Анализ на сложността на итеративни алгоритми

Тодор Дуков

Как анализираме един алгоритъм по сложност?

Нека започнем с един прост пример:

```
1  int find(int *array, int n, int value)
2  {
3      for (int i = 0; i < n; ++i)
4      {
5          if (array[i] == value)
6              return i;
7      }
8
9      return -1;
10 }
```

Да кажем, че искаме да проверим броят на инструкциите, която тази функция ще изпълни, преди да приключи работата си. Точен отговор не може да се даде. В зависимост от това къде се намира `value` във `array`, алгоритъма може да приключи много бързо или много бавно. Нещо, което обаче можем да направим, е да дадем горна и долна граница за бързодействието.
Ако `value` се намира в началото, то единственото което ще сме направили, ще са следните 4 операции:

- да инициализираме променливата `i` със 0
- да проверим верността на `i < n`
- да проверим верността на `array[i] == value`
- да върнем `i`

Нека сега да помислим какво ще стане в най-лошият случай (обикновено от тези ще се интересуваме) – `value` не участва в `array`. Тогава n на брой пъти ще изпълним следните 3 операции:

- проверяваме верността на `i < n`
- проверяваме верността на `array[i] == value`
- инкрементираме `i`

Освен тези $3n$ (където под n имаме предвид точно `n` от кода) операции, преди всичко трябва да инициализираме променливата `i` със 0, да се направи последната проверка на верността на `i < n` (която ще ни изкара от цикъла), и да върнем -1. Общо излизат $3n + 3$ операции.
Така виждаме, че в зависимост от входните данни, алгоритъма приключва работа за поне 4 стъпки и най-много $3n + 3$ стъпки. Такъв алгоритъм ще казваме, че има сложност по време $O(n)$. Разбира се, няма да е грешно и да кажем, че алгоритъма има сложност по време $\Omega(1)$, но това не ни дава никаква информация, защото всеки алгоритъм има такава сложност. Също така, понеже не използваме допълнителни променливи, алгоритъма ни има сложност по памет $\Theta(1)$.

Предимствата и недостатъците на този вид анализ

Най-голямото предимство на асимптотичния анализ, е неговата простота. Вместо да влачим някакви константни множители и събираеми, имаме колкото се може по-проста формула, която да описва сложността на нашия алгоритъм. Това дали един алгоритъм работи със две или три стъпки по-бързо/бавно не ни интересува особено много. При много голям вход те ще работят практически еднакво. В някакъв смисъл това ни помага да виждаме по-голямата картинка. Един алгоритъм може да бъде по-бърз от друг, но от по-бърз алгоритъм до по-бърз алгоритъм има голяма разлика.

Нека вземем за пример следната таблица:

n	$f(n) = \lceil \log_2(n) \rceil$	$f(n) = n$	$f(n) = n^2$	$f(n) = 2^n$
1	0	1	1	2
10	4	10	100	1024
100	7	100	10000	1267650600228229401496703205376
10000	13	10000	100000000	число със 3011 цифри
1000000	20	1000000	1000000000000	число със 301030 цифри

Алгоритъм със сложността n^2 ще е по-бавен от алгоритъм със сложност n, обаче скока в бързината е много по-малък от този между 2^n и n^2.
Този подход обаче си има своите недостатъци. Нека разгледаме два алгоритъма със сложности по време съответно n и $2^{2^{2^{1024}}}$. Ние ведната ще се втурнем да кажем, че първият алгоритъм е по-лош. Той е с линейна сложност, а вторият алгоритъм има константна сложност. Обаче преди вторият алгоритъм даде отговор, всички звезди ще умрат т.е. няма да доживем да чуем този отговор. Разбира се, от някъде нататък, за много голями входни данни, първият алгоритъм наистина ще работи по-бавно, но ние никога няма да работим с толкова големи данни. Тогава на практика, първият алгоритъм е по-добър, нищо че асимптотично се води по-лош. На нас това няма да ни интересува в курса по ДАА.

Сложност по време на някои алгоритми

Нека видим сложността на алгоритъма Bubble sort:

```
1  void sort(int *array, int n)
2  {
3      for (int i = 0; i < n - 1; ++i)
4      {
5          for (int j = 0; j < n - i - 1; ++j)
6          {
7              if (array[j] > array[j + 1])
8              {
9                  int temp = array[j];
10                 array[j] = array[j + 1];
11                 array[j + 1] = temp;
12             }
13         }
14     }
15 }
```

В най-лошия случай сложността $T(n)$ на функцията `sort` е следната:

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i-1} 1 = \sum_{i=1}^{n-1} (n-i-1) = (n-2) + (n-3) + \cdots + 1 + 0 = \frac{(n-2)(n-1)}{2} = \frac{n^2}{2} - \frac{3n}{2} + 1 \asymp n^2$$

По принцип $T(n)$ трябва да е сума от 4, а не от 1, но такъв константен брой операции, дори и приложени неконстантен брой пъти, не влияят на асимптотичното поведение.
Нека сега разгледаме следният алгоритъм за степенуване:

```
1  int exp(int base, int power)
2  {
3      int result = 1;
4
5      while (power > 0)
6      {
7          if (power % 2 == 1)
8              result *= base;
9
10         power /= 2;
11         base *= base;
12     }
13
14     return result;
15 }
```

Той се възползва от простата идея, че за да сметнем да кажем 3^8, можем вместо 8 пъти да умножаваме числото 3, да представим 3^8 като $3^4 \cdot 3^4$. Тогава 3^4 можем да сметнем веднъж, и да го умножим със себе си. Пак можем да представим 3^4 като $3^2 \cdot 3^2$ и да пресметнем 3^2 само веднъж и да го умножим със себе си. Така при по-голяма стойност на `power` си спестяваме много работа. С уговорката, че умножението е атомарна операция, сложността по време $T(n)$ (n е стойността на `power`) на функцията `exp` е следната:

$$T(n) = \sum_{\substack{i=n \\ i \leftarrow \frac{i}{2}}}^{1} 1 = \underbrace{1 + \cdots + 1}_{\substack{\text{колкото пъти можем да делим целочислено } n \text{ на 2 преди да получим 0}}} = \underbrace{1 + \cdots + 1}_{\text{около } \log(n) \text{ пъти}} \asymp \log(n)$$

Задачи

Задача 1. Да се определи сложността по-време за функцията:

```
int f1(int n)
{
    int s = 0;

    for (int i = 0; i < n; ++i)
    {
        for (int j = 0; j < i; j *= 2)
        {
            s += i * j;
        }
    }

    return s;
}
```

Задача 2. Да се определи сложността по-време за функцията:

```
int f2(int n)
{
    int s = 0;

    for (int i = 0; i * i < n; i += 3)
    {
        s += f1(i);
    }

    return s;
}
```

Задача 3. Да се определи сложността по-време за функцията:

```
int f3(int n)
{
    int s = 0;

    for (int i = 0; i < n * n; ++i)
    {
        for (int j = 0; j < i; ++j)
        {
            if (j % 2 == 0) s += f1(n);
            else s += f2(n);
        }
    }

    return s;
}
```